Today:

- Regular Languages
- Regular Expressions
- FSM Implementations
- Finite State Transducers
A formal language is used to distinguish precisely what sequences are allowed

- expressed mathematically, often recursively

Three important definitions:

- alphabet ($\Sigma$) - a set of characters / symbols
- word ($w$) - a finite sequence of characters / symbols
- language ($L$) - a [possibly infinite] set of words

Parse a word $w$ to decide if it is in the language $L$

- Accept if $w$ is in $L$, Reject if not in $L$
Modelling Computation (Review)

To decide a language, use a finite state machine (FSM).

Rules of the Game:

- Finite number of states: one of them is the Start state; one or more are the Final states.
- The FSM reads one character at a time.
- Transitions are based solely on the current state and the next character.
- A missing transition defaults to the dead state, which is not a Final state.
- If the FSM ends in a final state, then: Accept
- Else: Reject

\[ \Sigma = \{a, b, c\} \]

\[ L = \{\text{all words that have substring } abc\} \]
Modelling Computation (Review)

To decide a language, use a finite state machine (FSM).

Rules of the Game:

- Finite number of states: one of them is the Start state; one or more are the Final states.
- The FSM reads one character at a time.
- Transitions are based solely on the current state and the next character.
- A missing transition defaults to the dead state, which is not a Final state.
- If the FSM ends in a final state, then: Accept
- Else: Reject

$L = \{ \text{all words that don’t have substring } abc \}$
A regular language can be decided by a FSM.

- If you complement a regular language, i.e., swap Accept ↔ Reject, the result is a regular language.
- Regular languages are closed under complement.

Regular languages are also closed under:

- union
- catenation
- Kleene star

Write them using regular expressions.
Regular Expressions

If $L_1$ and $L_2$ are two regular languages, then

3rd • $L_1 \mid L_2$ is their union, i.e., use a word from $L_1$ or a word from $L_2$

2nd • $L_1L_2$ is their catenation, i.e., use a word from $L_1$ followed by one from $L_2$

1st • $L_1^*$ is its Kleene closure, i.e., use 0 or more catenations of words from $L_1$

Examples:

• 0 or more b’s: $b^*$
• begins with a b: $b(a|b)^*$
• begins and ends with a b: $b(a|b)^*b$
• begins or ends with a b: $b(a|b)^* \mid (a|b)^*b$
• begins and ends with different: $\lambda \mid a(a|b)^*b \mid b(a|b)^*a$
• exactly 3 long: $(a|b)(a|b)(a|b)$ OR $(a|b)^3$
• has substring abc: $(a|b|c)^*abc(a|b|c)^*$
• even number of a’s: $b^* (ab*ab*)^*$

Stephen Kleene (Regular Language Guru)
FSM Implementation

Follow transitions in a simple loop.

Algorithm:

\[
\begin{align*}
\text{state} &\leftarrow \text{Start} \\
\text{while there is still input} \{ & \text{c} \leftarrow \text{next input symbol} \\
& \text{if transition(state, c) exists then} \\
& \quad \text{state} \leftarrow \text{transition(state, c)} \\
& \text{else} \\
& \quad \text{Reject (OR . . . state} \leftarrow \text{Dead)} \\
\} \\
& \text{if state is a Final state then Accept} \\
& \text{else Reject}
\end{align*}
\]

Reasonable Implementations:
- Table Method
- Case Method

\[\Sigma = \{0, 1, 2, \ldots, 9\}\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1-9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Begin w/ 0</td>
<td>Begin w/ 1-9</td>
</tr>
<tr>
<td>Begin w/ 0</td>
<td>Dead</td>
<td>Dead</td>
</tr>
<tr>
<td>Begin w/ 1-9</td>
<td>Begin w/ 1-9</td>
<td>Begin w/ 1-9</td>
</tr>
</tbody>
</table>
FSM Augmentation: Actions

While following a transition, perform an action

- place actions on transitions following a slash
- should compute a useful property of the word

E.g., What might be a useful property?

- the integer’s value
- A1: \( \text{val} = c - '0'; \)
- A2: \( \text{val} = 10*\text{val} + (c - '0'); \)

\( \Sigma = \{0, 1, 2, \ldots, 9\} \)
Another possible action: output

- need to add a special symbol for EOF (usually $\$\$)

Problem: Construct a FSM with output that reports the parity of a sequence of bits

- E.g., $1011 \rightarrow 1$, $11011 \rightarrow 0$, $\lambda \rightarrow 0$
Example: Block Reduction

Problem: Construct a FSM with output that reports the 0/1 blocks of a binary sequence

- E.g., \(111000010011100011\) → \(1010101\)

Strategy:

- Output the first of each block.
**Case Method**

**Algorithm:**

- Use a large if / else if / . . .
- Use a nested switch / case

```java
switch (state) {
    case Start:
        switch (c) {
            case '0':
                state = BeginWith0;
                break;
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                state = BeginWith1to9;
                break;
        }
    break;
    case BeginWith0:
        state = Dead;
        break;
    case BeginWith1to9:
        break;
    default:
        state = Dead;
        break;
}
```